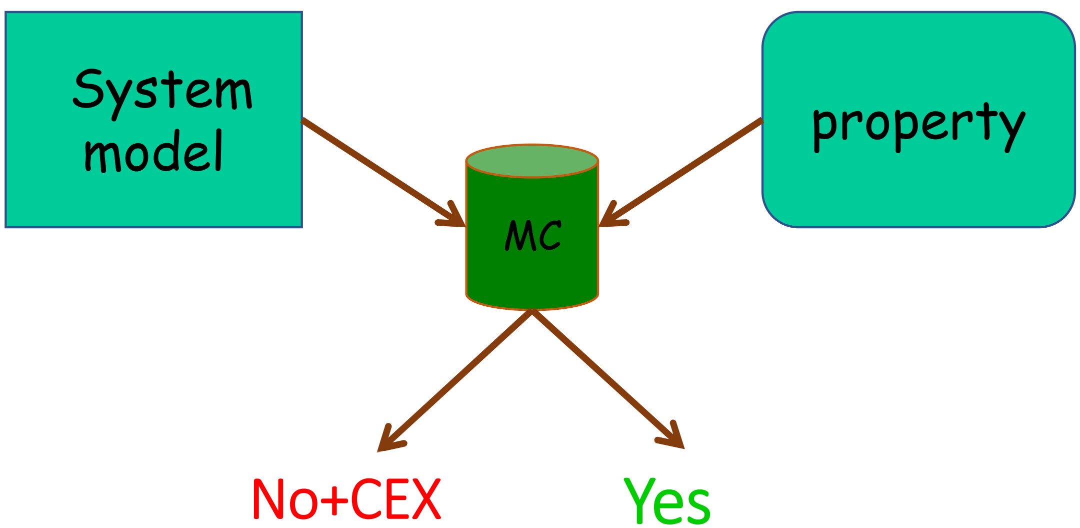
Automated Program Repair Using Formal Verification Techniques

Orna Grumberg
Technion, Israel

CREST Open Workshop (COW), London
January 20, 2020

Model Checking

- Given a system and a specification, does the system satisfy the specification.



Automated program repair

- Model checking finds bugs in the program
  - Bug: A program run that violates the specification
- Repair tool automatically suggests repair(s)
  - Repair: Changes to the program code, resulting in a correct program

In this talk – two approaches

- Exploit Model Checking technologies for program repair
  - Mutation-Based Program Repair
  - Assume, Guarantee or Repair

Sound and Complete Mutation-Based Program Repair

[Rothenberg, Grumberg, FM'16+work in progress]

Mutation-Based Program Repair

Sequential program

Assertions in code

Given set of mutations

Can we use these mutations to make all assertions hold?

Assignments, conditionals, loops and function calls

Assertion violation

operator replacement ($+ \rightarrow -$), constant manipulation ($c \rightarrow c + 1$)

Return all possible repairs

Example

```
int f(int x, int y){
        int z;
        if  (x + y > 8) {
                z = x + y;
        }  else {
                z = 9;
        }
        if (z ≥ 9)  z = z − 1;
        assert(z > 8);
        return z;
}
```

$x = 5, y = 2$

$z = 9$

$z = 8$

Example

```
   int f(int x, int y){
1.     int z;
2.     if (x + y > 8) {
3.         z = x + y;
4.     } else {
5.         z = 9;
6.     }
7.     if (z ≥ 9) z = z + 1;
8.     assert(z > 8);
9.     return z;
   }
```

At this point $z \geq 9$

Mutation list:
- Replace + with –
- Replace – with +
- Replace $>$ with $\geq$
- Replace $\geq$ with $>$

Note: Repairs are **minimal**

Repair list:

option 1:
- *line* 7: *replace* $\geq$ *with* $>$

option 2:
- *line* 7: *replace* $-$ *with* $+$

Example

```
int f(int x, int y){
    int z;
    if (x + y > 9) {
        z = x + y;
    } else {
        z = 10;
    }
    if (z ≥ 9) z = z − 1;
    assert(z > 8);
    return z;
}
```

At this point $z \geq 10$

Mutation list:

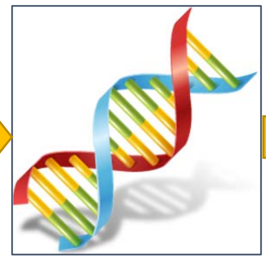
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >
- Increase constants by 1

Overview of our approach

```
int f(int x, int y){
1.    int z;
2.    if (x + y > 8) {
3.        z = x + y;
4.    } else {
5.        z = 9;
6.    }
7.    if (z ≥ 9)  z = z − 1;
8.    assert(z > 8);
9.    return z;
}
```

Input:
a buggy
program

Translation



Mutation

Repair

$line\ 7:\ replace\ operator\ \geq\ with >$
$line\ 7:\ replace\ operator\ -\ with +$
…

Output:
All **minimal** repairs,
sorted by size

Finding all correct programs / unsatisfiable constraint sets
from a finite set of programs / constraint sets

First step - Translation

Goal: Translate the program into a **set of constraints** which is **satisfiable iff the program has a bug**
(i.e. there exists an input for which an assertion fails)

Work by Clarke,Kroening,Lerda (TACAS 2004)
(CBMC)

- Simplification
- Unwinding of loops
  - a **bounded** number of unwinding
- Conversion to SSA

Correctness
is bounded

First step - Translation

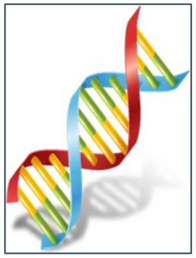
```
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z − 1;
    assert(z > 8);
    return z;
}
```

$$
\begin{aligned}
\{\ & g_1 = x_1 + y_1 > 8, \\
& \boldsymbol{z_2 = x_1 + y_1}, \\
& z_3 = 9, \\
& z_4 = g_1 ? z_2 : z_3, \\
& b_1 = z_4 \geq 9, \\
& z_5 = z_4 - 1, \\
& z_6 = b_1 ? z_5 : z_4, \\
& z_6 \leq 8 \\
\}
\end{aligned}
$$

First step - Translation

```
int f(int x, int y){
        int z;
        if  (x + y > 8) {
                z = x + y;
        } else {
                z = 9;
        }
        if (z ≥ 9)   z = z − 1;
        assert(z > 8);
        return z;
}
```

$$
\{\ g_1 = x_1 + y_1 > 8,\\
z_2 = x_1 + y_1,\\
\mathbf{z_3 = 9},\\
z_4 = g_1 ?\, z_2 : z_3,\\
b_1 = z_4 \geq 9\ ,\\
z_5 = z_4 - 1,\\
z_6 = b_1 ?\, z_5 : z_4,\\
z_6 \leq 8\\
\}
$$



Second step - Mutation

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >

$int\ f(int\ x, int\ y)\{$

1. $int\ z;$
2. $if\ \ (x + y > 8)\ \{$
3. $z = x - y;$
4. $\}\ else\ \{$
5. $z = 9;$
6. $\}$
7. $if\ (z \geq 9)\ \{$
   $z = z - 1;$
   $\}$
8. $assert(z > 8);$
9. $return\ z;$

$\}$

$g_1 = x_1 + y_1 > 8$

$\{\, z_2 = x_1 + y_1 , z_2 = x_1 - y_1 \}$

$\{\, z_3 = 9\}$

$z_4 = g_1 ? z_2 : z_3$

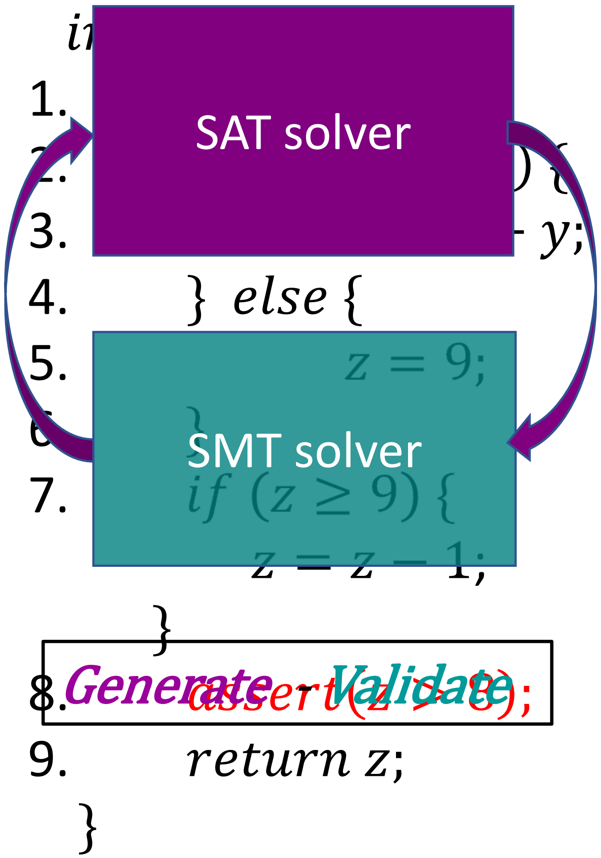
$\{\, b_1 = z_4 \geq 9 , b_1 = z_4 > 9\}$

$\{\, z_5 = z_4 - 1, z_5 = z_4 + 1\}$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

Third step - Repair



$\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \}$

$\{ z_3 = 9 \}$

$z_4 = g_1 ? z_2 : z_3$

$\{ b_1 = z_4 \geq 9 , b_1 = z_4 > 9 \}$

$\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \}$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

Making repair more efficient

Goal: reducing the search space

1. When a **correct mutated program** is generated (Validate succeeds)
   - Eliminate non-minimal correct mutated programs

2. When a **buggy mutated program** is generated (Validate fails)
   - Eliminate "similar" buggy mutated programs

Buggy mutated program

Unsuccessful repair:
A set of mutations M that results in a buggy program

Elimination:

- Find a small explanation S for the bug
  - S is a set of statements in the code
- Disallow any mutated program, containing S

Fault localization

Fault localization: A (small) explanation S to a bug

In most works:

- **May** explanation
  - Changes to statements from S **may** result in a repaired program

Fault localization

Fault localization: A (small) explanation S to a bug

In our work:

- **Must** explanation
  - If **none** of the statements in S is changed, then
    - regardless of changes applied to other statement
    - **the same bug will remain**
- **⇒ S must be changed**

Fault localization: example

$int\ f(int\ x, int\ y)\{$

1. $int\ z;$
2. $z = x$
3. $if\ \ (x >= \mathbf{0})\ \{$
4. $x = x + 1; \mathrm{y} = \mathrm{x} + 2;$
5. $\}\ else\ \{$
6. $z = 9;$

   $\}$

7. $assert(z > 0);$
8. $return\ z;$

$\}$

Fault localization: example

	`int f(int x, int y){`	erroneous run:
1.	`int z; int t;`	$x=0, y=0$
2.	`z = x`	$z=0$
3.	`    if (x >= 0) {`	
4.	`        x = x + 1; y = x + 2;`	$x=1, y=2$
5.	`    } else {`	
6.	`        z = 9;`	
	`    }`	
7.	`    assert(z > 0);`	$z=0$
8.	`    return z;`	
	`}`	

Repair: line 3 should change to (x > 0)

Fault localization by slicing

		execution slice	dynamic slice	our slice
	$int\ f(int\ x, int\ y)\{$			
1.	$int\ z;\ int\ t;$			
2.	$z = x$	•	•	•
3.	$if\ \ (x >= 0)\ \{$	•		•
4.	$\quad x = x + 1; y = 0;$	•		
5.	$\}\ else\ \{$			
6.	$\quad z = 9;$			
	$\}$			
7.	$assert(z > 0);$	•	•	•
8.	$return\ z;$			
	$\}$			

Theorem: Our algorithm is sound and complete

That is, no good repair is eliminated by our search space pruning

Summary

Mutation-based formal repair is not as scalable as test-based approaches

- It can assist a programmer in debugging in initial stages of development
  - When bugs are simple, but many
- It also can help beginner programmers
  - Educational tool for students (experiments on Codeflaw)
- Pruning has a significant impact in making the search for repair(s) efficient

Assume, Guarantee or Repair

[Frenkel, Grumberg, Pasareanu, Sheinvald, TACAS'2020]

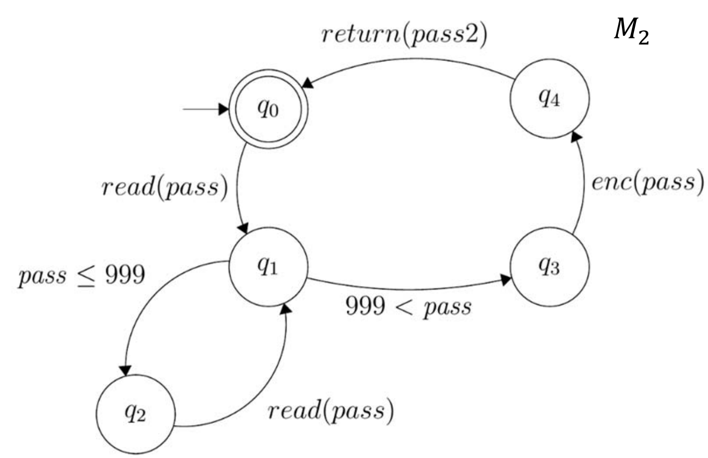
Goal

- Exploit the partition of the system into components
- Compositional model checking verifies small components and conclude the correctness of the full system
- If a vulnerability is found, **repair** is applied to one of the components

Communicating systems

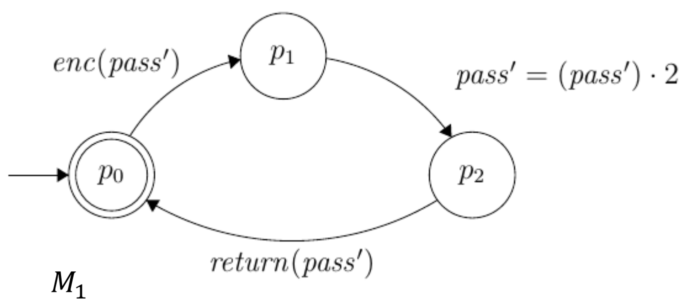
- C-like programs
- Described as a control-flow graph (automaton)
- Use automata learning algorithms

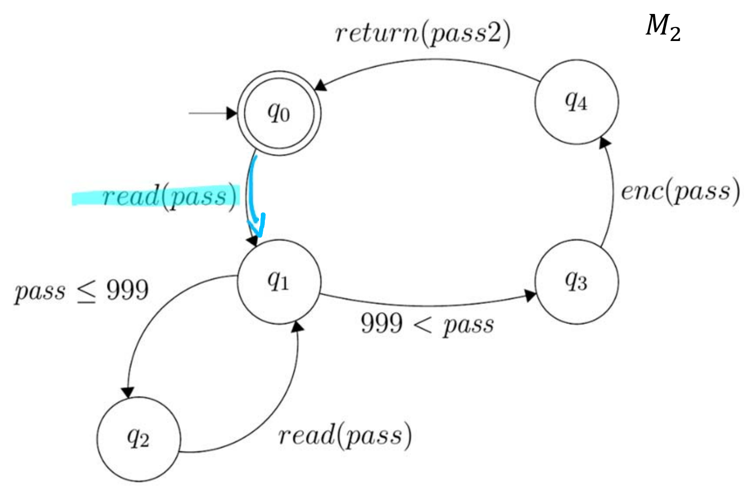
```
while (true)
   pass = readInput;
   while (pass ≤ 999)
    pass = readInput;
   pass2 = encrypt(pass);
   return pass2;
```



Example

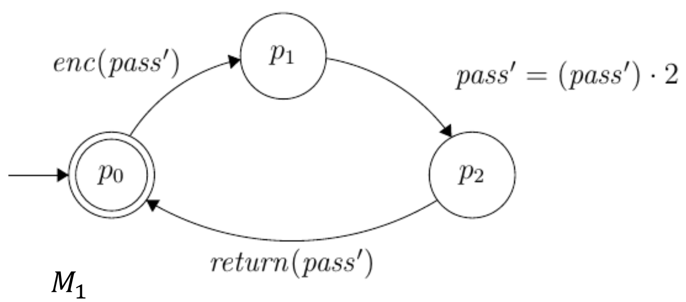
- Components synchronize over common channels

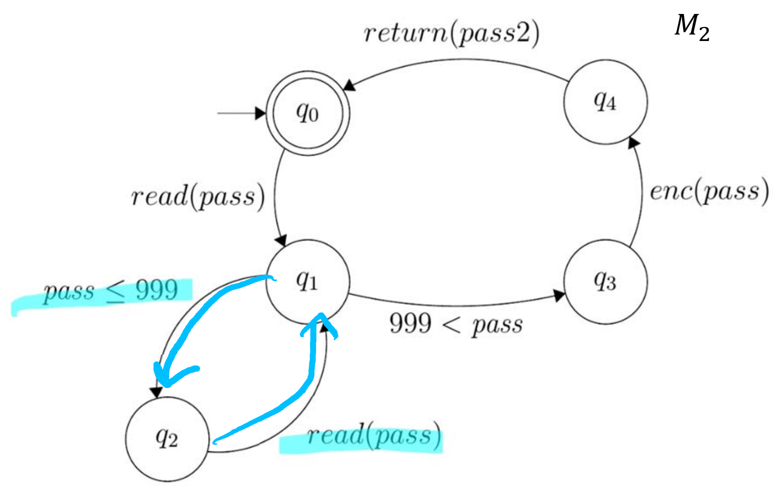




Example

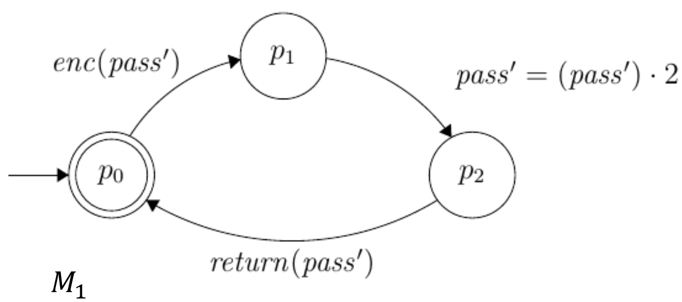
- Components synchronize over common channels

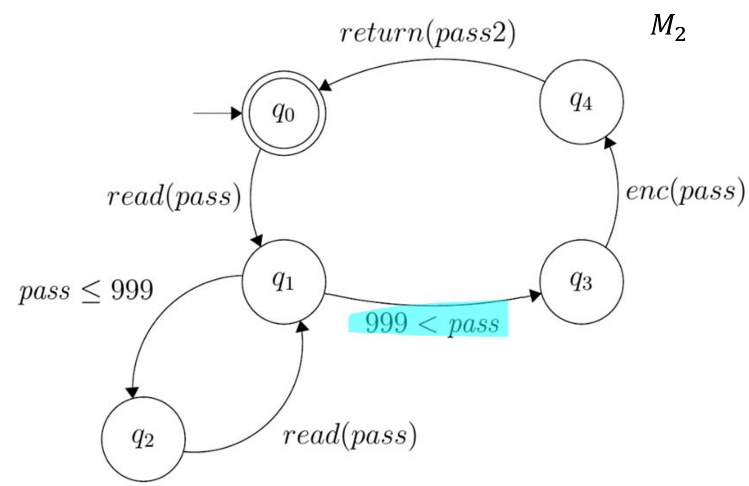




Example

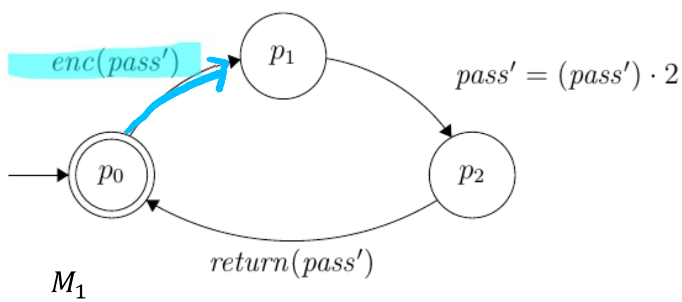
- Components synchronize over common channels

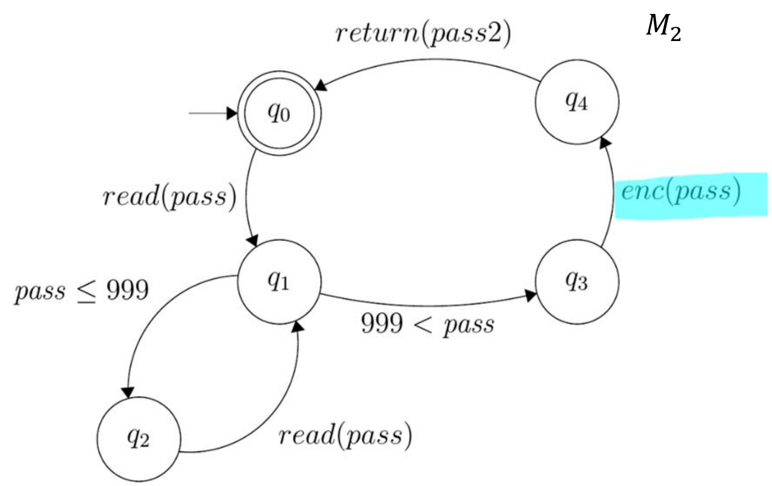




Example

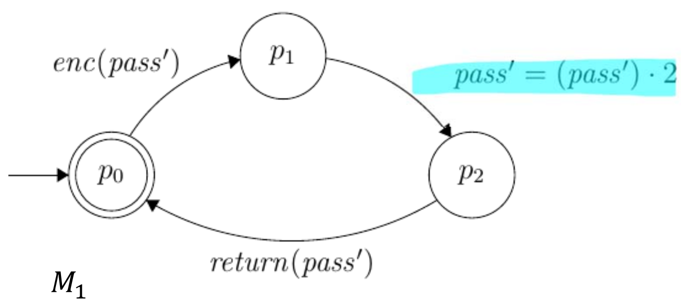
- Components synchronize over common channels

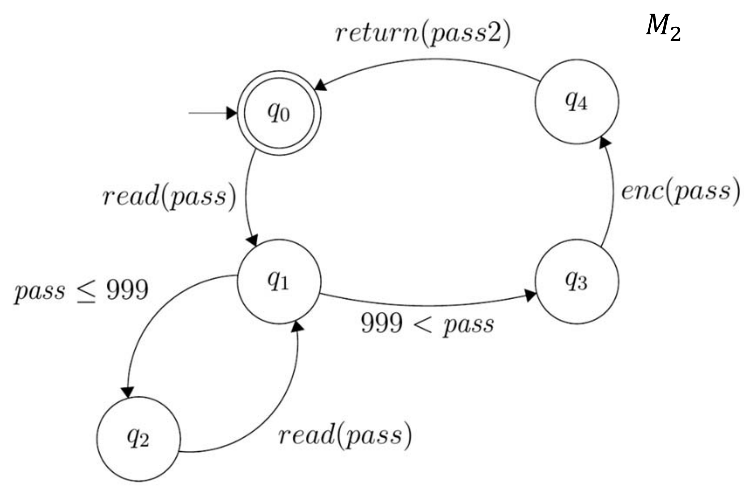




Example

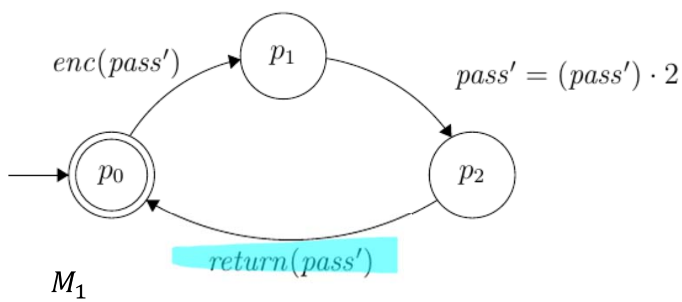
- Components synchronize over common channels

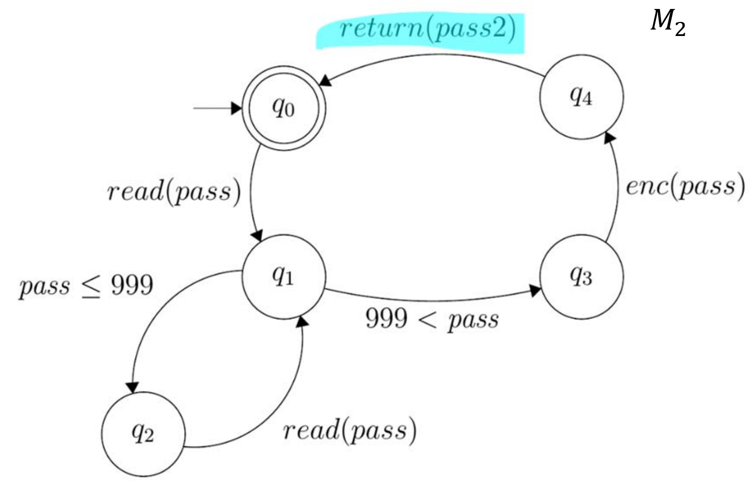




Example

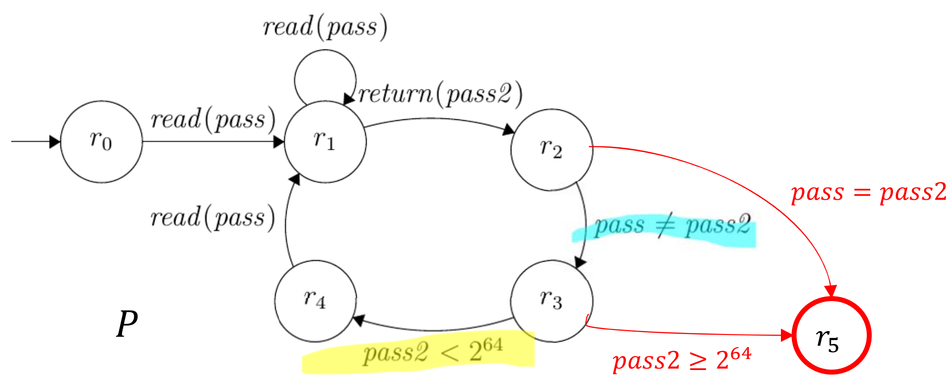
- Components synchronize over common channels





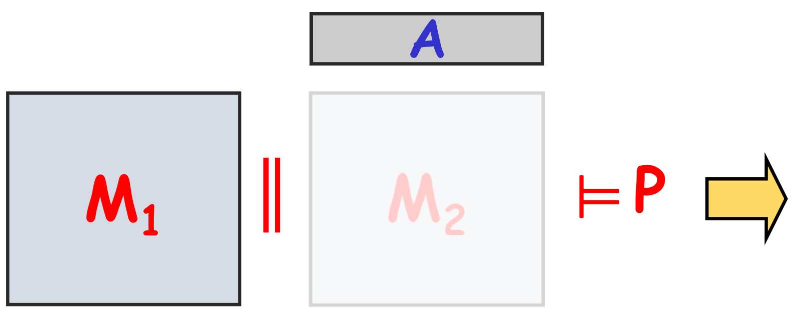
Specifications

- Safety requirements – given as an automaton
- Behavior of the program through time
- “the entered password is different from the encrypted password”
- “there is no overflow”



Assume-Guarantee (AG) Rule

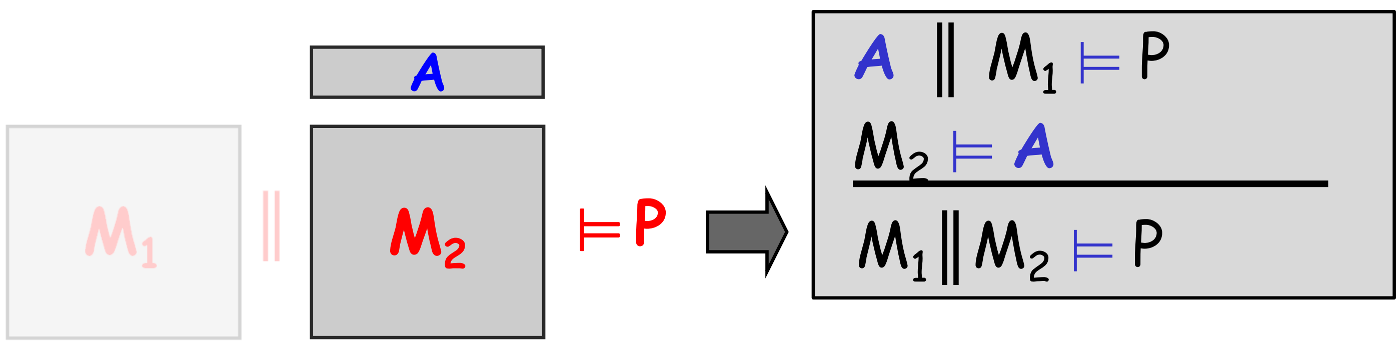
1. check if a component M_1 **guarantees** P when it is a part of a system satisfying **assumption** A



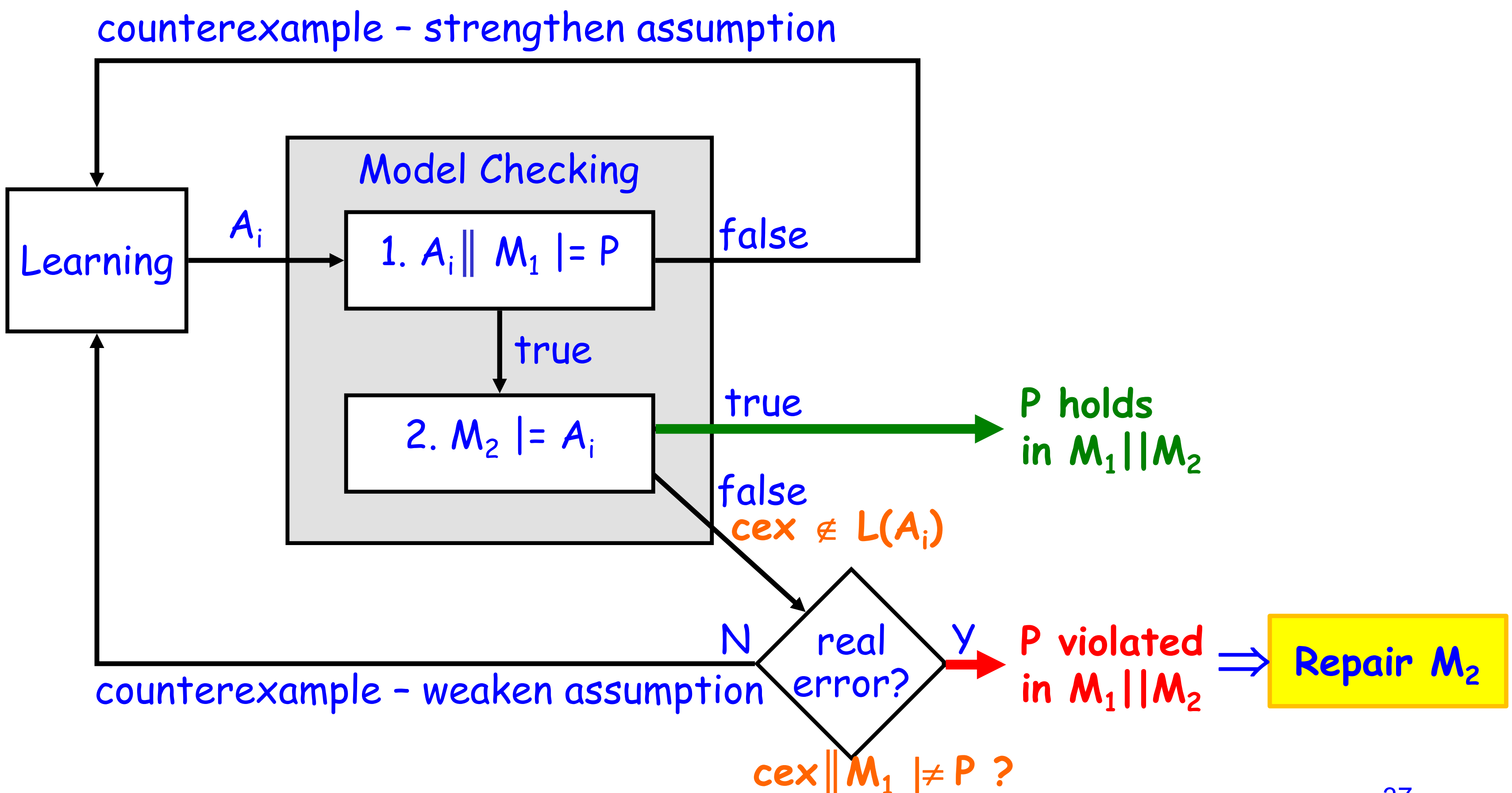
$$\frac{A \parallel M_1 \models P}{M_1 \parallel M_2 \models P}$$

Assume-Guarantee (AG) Rule

1. check if a component M_1 **guarantees** P when it is a part of a system satisfying **assumption** A
2. show that the other component M_2 (the environment) satisfies A



Assume Guarantee or Repair



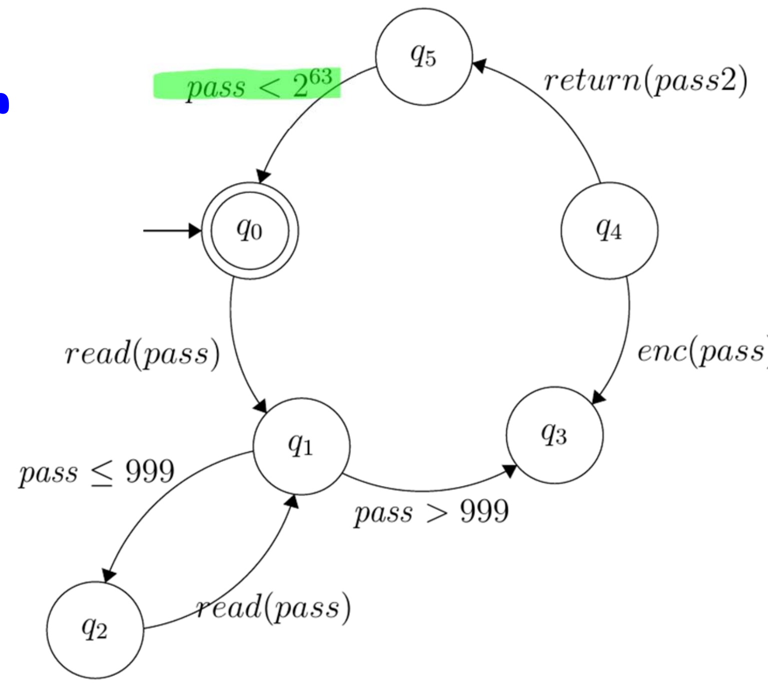
Semantic Repair

- The counterexample contains constraint **-- violating overflow**
- learn a constraint $\mathcal{C}$ such that:
- $\boldsymbol{\mathcal{C}} \wedge pass > 999 \wedge pass2 = pass \cdot 2 \wedge \mathbf{pass2} \geq \mathbf{2^{64}} \rightarrow false$
- $\boldsymbol{\mathcal{C}}$ is over the input variables - $pass$.

- $\mathcal{C} \coloneqq \forall pass2 : pass > 999 \wedge pass2 = pass \cdot 2 \wedge pass2 \geq 2^{64} \rightarrow false$
- After quantifier elimination & simplification: $\boldsymbol{\mathcal{C} = pass < 2^{63}}$

Abduction - "Logical Magic"

Repair




```
while (true)
   pass = readInput;
   while (pass ≤ 999 or pass≥ 2^63)
    pass = readInput;
   pass2 = encrypt(pass);
   return pass2;
```

Syntactic repair

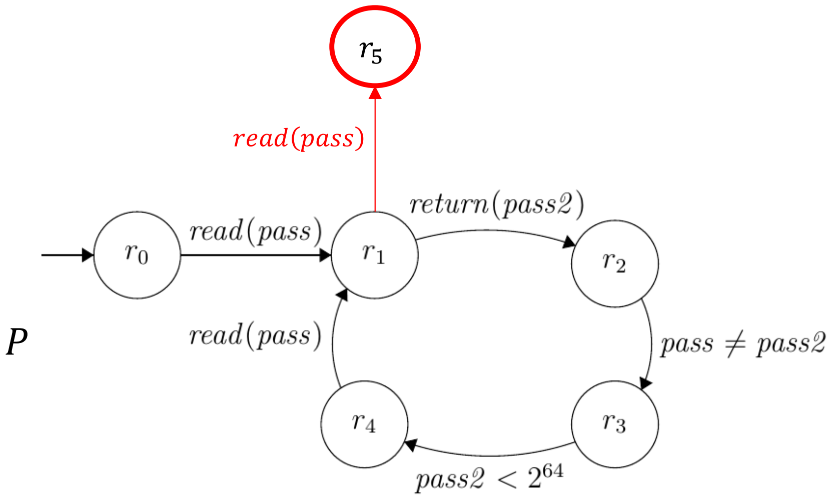
- The counterexample t contains no constraint
  - It consists of communication actions and assignments
- Abduction will not help

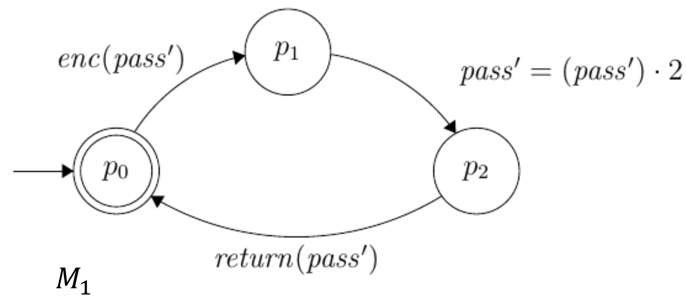
3 methods to removing counterexample t:

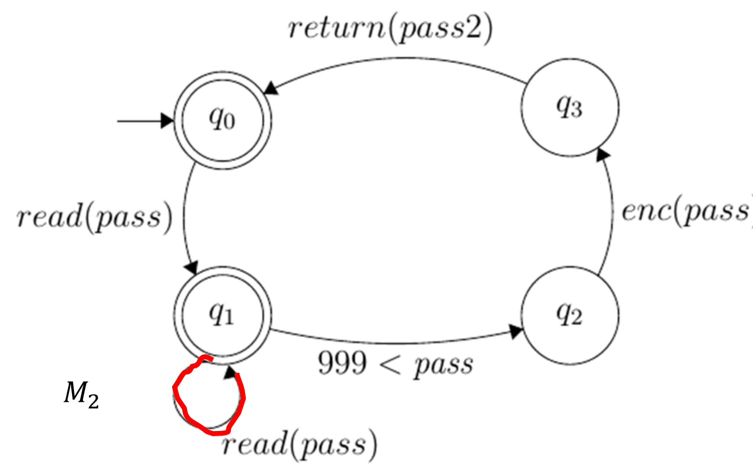
- Exact: remove exactly t from M_2
- Approximate:
- Aggressive:

Example – Syntactic Repair

No self loop, cannot *read* more than once each time!



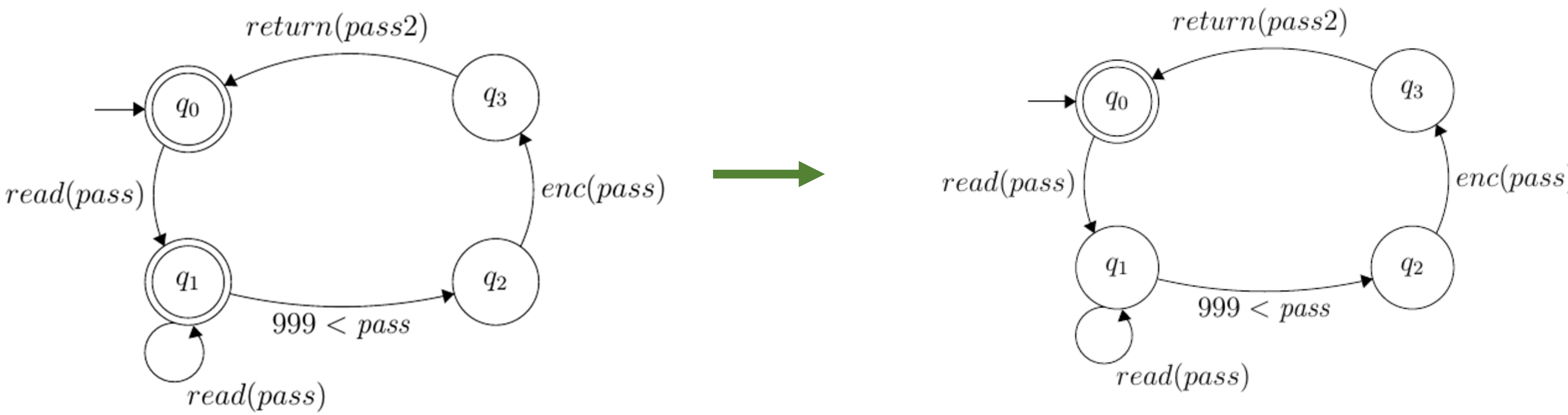




Multiple reads are allowed

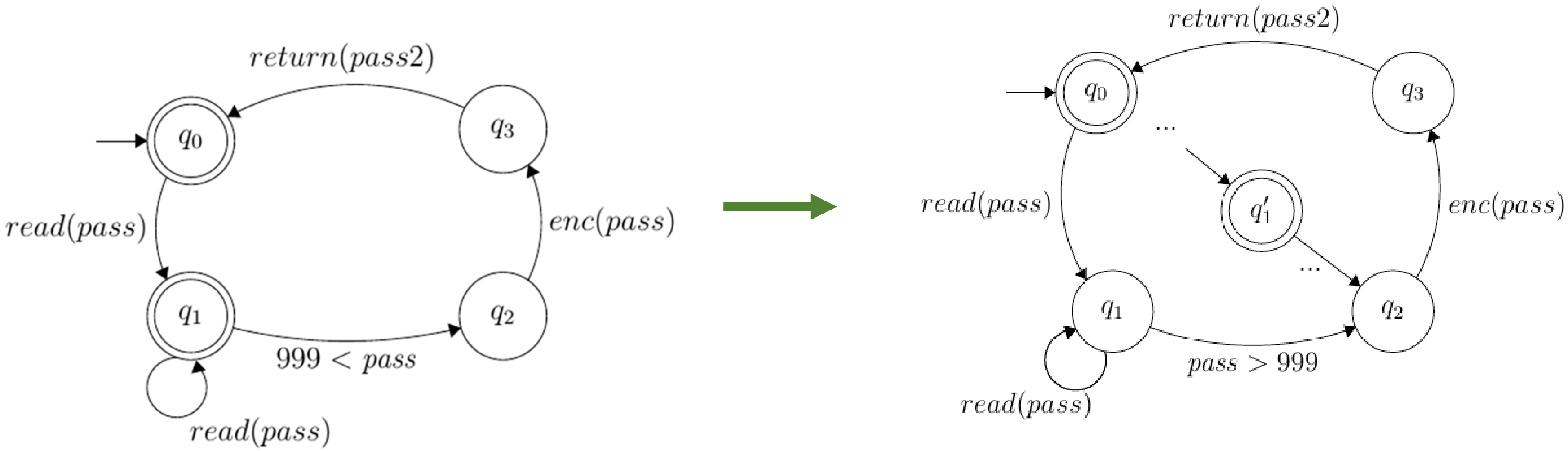
Agressive Repair

- Remove accepting states (can make the language of M_2 empty)



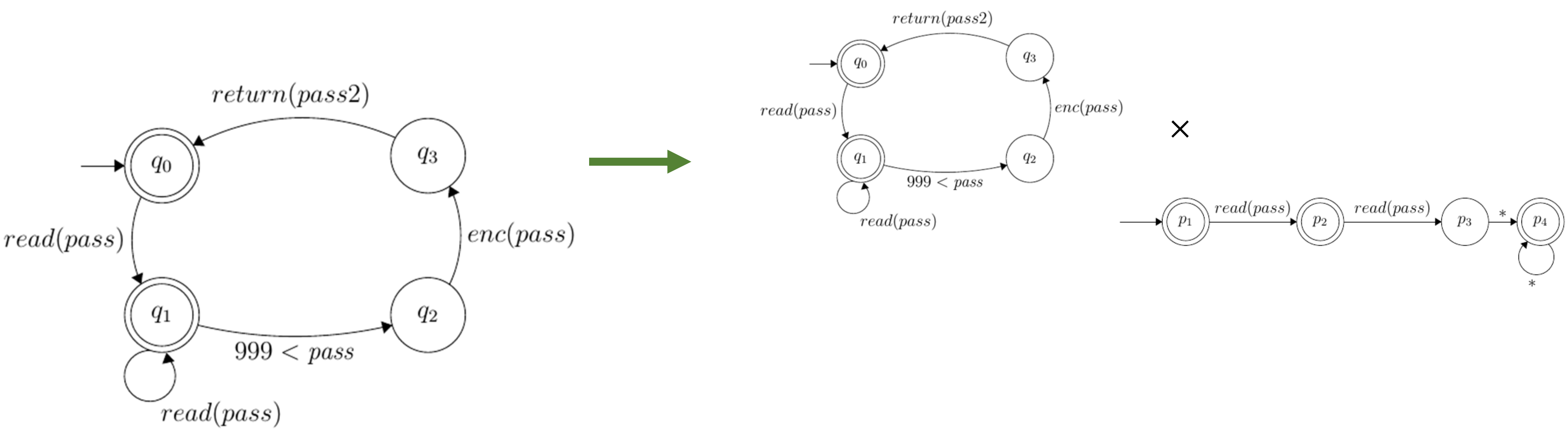
Approximate Repair

- Add an intermidiate state to eliminate bad traces

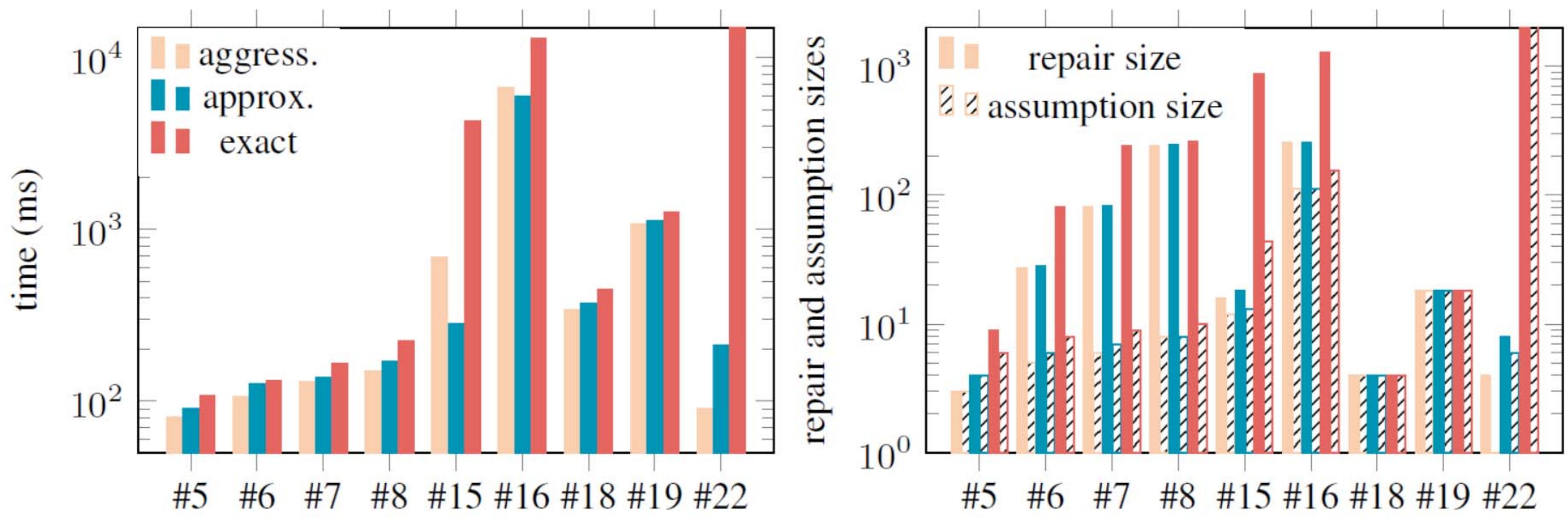


Exact Rapair

- Remove bad traces one by one
- First bad trace spotted is $read(pass),\ read(pass)$



Comparing Repair Methods (logarithmic scale)



#15, #16, #18, #19 apply also abduction

Summary

- Learning-based Assume Guarantee algorithm for infinite-state communicating programs
- Semantic and syntactic repair
- Experiments provide proof of concept

Summary

- Two approaches to automatic program repair
  - based on formal method technologies

Thank you